Observing the Uptake of a Language Change Making Strings Immutable

Manuel Maarek
Heriot-Watt University, Edinburgh, Scotland, UK
m.maarek@hw.ac.uk
https://orcid.org/0000-0001-6233-6341

Abstract

To address security concerns, a major change was introduced to the OCaml language and compiler which made strings immutable and introduced array of bytes as replacement for mutable strings. The change is progressively being pushed so that ultimately strings will be immutable. We have investigated the way OCaml package developers undertook the change. In this paper we report on a preliminary observation of software code from the main OCaml package management system. For this purpose we instrumented versions of the OCaml compiler to get precise information into the uptake of safe strings.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases software evolution, programming language evaluation, immutability, secure programming

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.6

1 Introduction

Following the LaFoSec study [6], a major change was introduced to the OCaml language and compiler which splits the `string` data type into its now immutable version and a new `bytes` data type representing arrays of bytes. The change is gradual and was initially made in version 4.02.0 (see Table 1). The new immutable strings were available in the language and standard library as well as the new array of bytes but the compiler would initially still allow to mutate strings, issuing a warning to the developer. Immutable strings are now default unless a specific option is set. In future versions of the compiler strings will only be immutable.

The security rationale for the change was that any part of a code could alter a string value it has access to regardless of encapsulation (e.g. string literals of the standard library could be modified). The importance of the change and the fact the change is being carried out gradually to retain backward compatibility during a period of time makes it an interesting case to evaluate how developers perceive such changes and how they implement it within their own development. As a starting point, we want to mechanically inspect the uptake within developers' codes. We have therefor developed versions of the compiler that observe the uses of `string`, `bytes`, their associated operations, and the relevant compiler options. We then run our observation compilers on openly accessible OCaml code. We chose to mine the OCaml code that are provided through OCaml's main package management system OPAM. In this paper, we present the experiment setting and discuss our findings.

Table 1 OCaml versions significant for string immutability.

	OCaml version	OPAM version	Number of packages
Mutable strings	**4.01.0** (2013-09)	1.1.2 (2014-06)	1382
Introduction of safe strings	4.02.0 (2014-08)		
Last version of the 4.02 family	**4.02.3** (2015-08)	1.2.1 (2015-03)	1699
Safe string as default	4.06.0 (2017-11)		
Latest version	**4.07.0** (2018-07)	1.2.2 (2015-04)	1921

Plan

In Section 2 we give more details about immutability, OCaml and LaFoSec which form the context of this research. We present our experimental setting in Section 3 and the outcome of our observations in Section 4. We finally conclude and discuss perspectives for this work in Section 5.

2 Context

In this section we give the context to this research where we planned an observation of the uptake of a security-induced change in the OCaml language.

2.1 Immutability

Mutability is a key feature of most programming language as it is a way for programmer to represent the change of state of the system they develop. Some programming languages such as functional languages prevent variable mutation, the state of the system is represented by the arguments passed from operations to operations following the execution flows of the program. An advantage of immutability is that it enables referential transparency which means that a variable will always represent the same value within its scope: passing the variable as argument to other operations or concurrent operations cannot modify its value. As a result immutability is more secure as it brings certainty to the programmer. Mutability is a source of vulnerability as the following CWEs (Common Weakness Enumeration) exemplify: *CWE-374: Passing Mutable Objects to an Untrusted Method*[1], *CWE-471: Modification of Assumed-Immutable Data (MAID)*[2]. Immutability is increasingly available in programming languages although immutability could cause usability challenges for developers used to state-base programming paradigms [11, 3].

2.2 OCaml

OCaml[3] is a functional language. While being a functional language, it offers some mutable data structures and variable references. In OCaml, strings are mutable since the early versions of the language. This design choice seems to have been made to allow for easy use of strings as byte arrays where mutability is essential.

[1] `https://cwe.mitre.org/data/definitions/374.html`
[2] `https://cwe.mitre.org/data/definitions/471.html`
[3] `https://ocaml.org/`

OCaml is also a strongly typed programming which means that each expression is associated to a unique type. Its inference system implemented with its type system in the compiler, makes it unnecessary in most cases to annotate the source code with type information.

2.3 LaFoSec study

The LaFoSec study [6] analysed the intrinsic security properties of functional languages and OCaml in particular. It was preceded by a study on Java Security [5]. LaFoSec detailed the features such as encapsulation which are essential for security development, and identified ways in which they could be bypassed. It highlighted the security issue of immutable strings. Let us consider the following example of a function returning the days of the week as strings (using OCaml 4.01.0 version).

```
# let weekday n =
      match n with
      | 1 -> "Monday"
      | ...
val weekday : int -> string = <fun>
# weekday 1
- : string = "Monday"
```

The string literal of the function could be changed by any piece of code executed outside of the current module, e.g. by any code dynamically loaded.

```
# let s = weekday 1; s.[0] <- 'F'; s.[1] <- 'r'; s.[2] <- 'i';
# weekday 1
- : string = "Friday"
```

To prevent such tempering, LaFoSec recommended to protect string literals by wrapping them in a call to the `String.copy` function, which means that each call to `weekday` will issue a new copy of the string literal. In addition, LaFoSec recommended to define an abstract module wrapping strings and which only offers access operations that do not mutate strings. These solutions were implemented in the secure XML validator prototype developed as part of LaFoSec and presented in [4]. The issue the mutable strings could cause to the reliability of higher level formal development is discussed in [1], highlighting that one can temper with the output of a logical frameworks. And [2] is another example of a formal development which required to rely on a custom string library for the system to retain security guarantees.

2.4 Change

The change introduced by the OCaml development team makes the values of the type `string` immutable by default and introduce a new type `bytes` for byte arrays (distinguishing the two common uses of strings). The change could imply additional operations and therefor a performance cost when needing to use `string` operations on `bytes` values as discussed in [10].

The change means that the methods of the `String` module (of the standard library) which were modifying strings in-place are now deprecated. When used, they raise a warning or an error if the `-safe-string` option of the compiler is used. Here is an example of warning issued by the 4.07.0 compiler when calling `String.set`.

```
# String.set;;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
- : bytes -> int -> char -> unit = <fun>
```

Note that the `String.set` function, which remained for backward compatibility, is an now alias to `Bytes.set` and expect as first argument a value of type bytes unless in unsafe mode. Calling `String.set` with a value of type `string` raises by default a type error with OCaml 4.07.0 version (OCaml 4.02.3 version would allow it by default).

```
# String.set "Monday";;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
Error: This expression has type string but an expression was expected
         of type bytes
```

The functions that are now deprecated following the change are `String.set` (which corresponds to the `s.[0] <- 'F'` syntax sugaring notation), `String.create`, `String.copy`, `String.fill`. Note that `unsafe_*` variants of `String` methods existed and are still available. E.g., `String.unsafe_get` allows to access a character of a string by its index as `String.get` does, but does it without runtime verification that the access is done within the boundaries of the string (this is used for optimisation purposes).

3 Experimental Setup

Instrumented compilers

To observe the manner OCaml users have adopted the new immutable strings and its mutable counterpart, we created variants of the OCaml compiler that records specific usage information. These instrumented compilers record the compiling options, the occurrence of expressions of type `string` and `bytes`, and the occurrence of calls to string and bytes methods[4].

Working within the compiler means that we have certainty about the information we gather: type information has been inferred and checked by the compiler. An alternative would have been to work at the source level where similar content could have different forms (e.g. qualified forms vs. unqualified forms), where some information are not available (e.g. type information, command line options), and noise needs to be filtered out (e.g. commented code).

We developed three instrumented compilers (or observer compilers) for the OCaml versions we listed in Table 1. We refer to the compiler for OCaml 4.01.0 version as `4.01.0` (resp. `4.02.3`, `4.07.0`) and to our instrumented compiler as `4.01.0+obs` (resp. `4.02.3+obs`, `4.07.0+obs`). Note that the three versions of OCaml are not retrocompatible and therefore require dedicated compilers.

OCaml being a compiled and strongly typed language, a compilation is necessary to execute OCaml programs and the compilation process includes typing the program after parsing and before code generation. The instrumentation we created are placed at the end of the typing when a typed Abstract Syntax Tree (AST) of the program is available.

Note that our current instrumented compilers look for `string`/`bytes` expressions and `string`/`bytes` methods without investing further the context of use or within the program flows. We are considering making such finer grained investigation in later research.

[4] Note that some of the information we collected could be obtained with the `-annot` option but would have required additional processing.

Code repository

We also decided to work within OCaml's OPAM package management system[5] as source codes in such system come alongside compiling instructions. This means that we were able to automate the process of gathering information about individual packages and files. While using OPAM means that we could easily access compilation commands, it also implies that the targeted audience are package developers who are more likely to be professionals or experts rather than general users of OCaml.

OPAM makes it trivial to deploy such custom compiler and retaining the dependency of packages of the original version.

Related work on code mining

A number of related works have been mining openly available code [9], or have investigated version histories to observe change [12, 8], sometimes working on the AST rather than source code [7].

4 Results of Observations

We identified packages that were available for each of the three versions of OCaml we wanted to base our research on (see Table 1) highlighting stages of the gradual change. The number of commonly available packages is 353. We have therefore used our observation versions of the compilers to install these 353 packages as we were interested in seeing the changes developer made since the introduction of immutable strings. Although we installed the same packages for each compiler version, or more precisely the versions of the same packages that corresponded to each compiler version, the number of files that were compiled successfully varied depending of the version. Using the default compilers (`4.01.0`, `4.02.3`, `4.07.0`), some compilations failed. This was in particular the case for `4.02.3` which could be explained by the fact that 4.02 family saw a number of minor versions making packages maintenance less effective. A package failing to compile meant that others could not be compiled due to missing dependency. A small number of compilations which compiled successfully with the default compiler failed or stalled with our instrumented compilers (`4.01.0+obs`, `4.02.3+obs`, `4.07.0+obs`). This was due to the instrumentation we designed with a compiler module to iterate over the OCaml AST which would sometimes reach the stack limit. As Table 2 shows, less files compiled successfully with version 4.02.3.

Table 2 Number of files compiled per version.

Version	Number of packages installed successfully	Number of files compiled successfully
`4.01.0`	212	
`4.01.0+obs`	201	7211
`4.02.3`	46	
`4.02.3+obs`	38	2253
`4.07.0`	224	
`4.07.0+obs`	221	10032

[5] `https://opam.ocaml.org/`

Use of unsafe string compiler option

We tracked the use of compiler options. This is only applicable to versions of the compiler released after the change. The gradual transition meant that there was limited incentive for the developers to use the safe compilation option in 4.02.3 version while 4.07.0 version does. Table 3 shows that such change of behaviour by the compiler were followed by an uptake of safer compiler options.

Table 3 Use of unsafe compiler option.

Version	Package use of unsafe option	File compiled with unsafe option
`4.01.0+obs`	NA / 201	NA / 7211
`4.02.3+obs`	30 / 38	2100 / 2253
`4.07.0+obs`	0 / 221	0 / 10032

Expressions of type `string` and `bytes`

The observation compilers also counted the number of expressions of type `string` and `bytes` as well as the overall number of expressions in the files that compiled successfully. This information gives an estimate of the use of these types of expression although it did not include functions with these types as either argument or output. Table 4 shows a slight decrease of the number of `bytes` expressions in proportion to `string` expressions.

Table 4 Expressions of type `string`/`bytes`.

Version	`string` expressions	`bytes` expressions	Ratio	Total number of expressions
`4.01.0+obs`	732390	NA	NA	7332863
`4.02.3+obs`	205165	3335	0.0162	2197761
`4.07.0+obs`	864876	13466	0.0155	9989802

Calls to unsafe `String` functions

The observation compilers record the usage of unsafe functions from the `String` module of the standard library. This recording is not transitive, so if a call is made to a function which is an alias to a unsafe `String` function, this will not be identified. Table 5 shows a decrease in the number of calls to unsafe `String` functions, although for 4.02.3 version the decrease could also be due to the set of packages that compiled rather than being due to the introduction of the change.

Table 5 Call to unsafe `String` functions.

Version	Packages with call to unsafe functions	Files with call to unsafe functions
`4.01.0+obs`	43 / 201	153 / 7211 (2.12%)
`4.02.3+obs`	5 / 38	16 / 2253 (0.71%)
`4.07.0+obs`	2 / 221	4 / 10032 (0.03%)

Note that the few remaining cases of call to unsafe `String` methods with 4.07.0 version are the consequence of syntax sugaring. The syntax sugaring notations to get (`s.[2]`) or set

(`s.[2] <- 'i'`) a character are respectively translated into the following calls `String.get s` and `String.set s 2 'i'`. This is still the case for 4.07.0 version which results in the type checker issuing a warning when identifying the `String.set` method and accepting its call when the argument is of type `bytes` (see an illustration with the following two snippets of code). This explains why for 4.07.0 version, the cases of call to `String.set` are compatible with the the absence of use of unsafe string compiler option.

```
# let s = "Monday";;
val s : string = "Monday"
# s.[2] <- 'i';;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
Error: This expression has type string but an expression was expected
         of type bytes
```

```
# let b = Bytes.of_string "Monday";;
val b : bytes = Bytes.of_string "Monday"
# b.[2] <- 'i';;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
- : unit = ()
```

5 Conclusion and Future Work

The purpose of our research is to understand how software developers refactor their code in light of security focused language changes. In this paper we presented our compiler instrumentation and initial findings into the uptake of a gradual change introduced in the OCaml language and compiler to make strings immutable. We relied on the OCaml type system and package management system to create instrumented compilers collecting accurate information about the uptake. Although the study only looked at three versions of the compiler, it shows a good uptake by the OCaml package developers.

As this is an initial experiment there are many ways it could be improved and expanded. We aim to apply the same strategy to more versions of the compiler to get a clearer picture of the uptake. We also would like to expand the investigation by looking at more diverse repositories of code, and comparing with similar changes in other programming languages.

We have not investigate in details how individual development proceeded with their change and would like to do so to categories these strategies. It would be interesting to combine this code based investigation with the way the change and strategy were perceived by OCaml developers. This could be the base to tailor such changes to optimise uptake or to create adequate interventions to drive changes.

References

1 Mark Adams. Flyspecking Flyspeck. In *Mathematical Software – ICMS 2014*, Lecture Notes in Computer Science, pages 16–20. Springer, Berlin, Heidelberg, August 2014. doi:10.1007/978-3-662-44199-2_3.

2 David Cadé and Bruno Blanchet. Proved Generation of Implementations from Computationally Secure Protocol Specifications1. *Journal of Computer Security*, 23(3):331–402, January 2015. doi:10.3233/JCS-150524.

3 M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. Glacier: Transitive Class Immutability for Java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506, May 2017. doi:10.1109/ICSE.2017.52.

4 D. Doligez, C. Faure, T. Hardin, and M. Maarek. Avoiding Security Pitfalls with Functional Programming: A Report on the Development of a Secure XML Validator. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 209–218, May 2015. doi:10.1109/ICSE.2015.149.

5 Éric Jaeger, Olivier Levillain, and Pierre Chifflier. Mind Your Language (s) – A Discussion about Languages and Security (Long Version). In *First Workshop on Language-Theoretic Security (LangSec) at the IEEE CS Security & Privacy Workshops*, 2014.

6 LaFoSec. Security and Functional Languages (Étude de La Sécurité Intrinsèque Des Langages Fonctionnels). Technical report, ANSSI (National Cybersecurity Agency of France), Main authors: D. Doligez, C. Faure, T. Hardin, M. Maarek, 2011.

7 M. Martinez, L. Duchien, and M. Monperrus. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In *2013 IEEE International Conference on Software Maintenance*, pages 388–391, September 2013. doi:10.1109/ICSM.2013.54.

8 O. Meqdadi, N. Alhindawi, M. L. Collard, and J. I. Maletic. Towards Understanding Large-Scale Adaptive Changes from Version Histories. In *2013 IEEE International Conference on Software Maintenance*, pages 416–419, September 2013. doi:10.1109/ICSM.2013.61.

9 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM. doi:10.1145/2635868.2635922.

10 Gerd Stolpmann. Immutable Strings in OCaml-4.02 (Blog on Camlcity.Org). http://blog.camlcity.org/blog/bytes1.html, July 2014.

11 S. Weber, M. Coblenz, B. Myers, J. Aldrich, and J. Sunshine. Empirical Studies on the Security and Usability Impact of Immutability. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 50–53, September 2017. doi:10.1109/SecDev.2017.21.

12 T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005. doi:10.1109/TSE.2005.72.